FORTRAN SUBROUTINES FOR COMPUTING APPROXIMATE SOLUTIONS OF WEIGHTED MAX-SAT PROBLEMS USING GRASP

MAURICIO G.C. RESENDE, LEONIDAS S. PITSOULIS, AND PANOS M. PARDALOS

ABSTRACT. This paper describes Fortran subroutines for computing approximate solutions to the weighted MAX-SAT problem using a greedy randomized adaptive search procedure (GRASP). The algorithm [Resende, Pitsoulis, and Pardalos, 1997] is briefly outlined and its implementation is discussed. Usage of the subroutines is considered in detail. The subroutines are tested on a set of test problems, illustrating the tradeoff between running time and solution quality.

1. INTRODUCTION

Let $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_m$ be m clauses, involving n Boolean variables $x_1, x_2, \ldots, x_n$, which can take on only the values `true` or `false` (1 or 0). In addition, for each clause $\mathcal{C}_i$, there is an associated nonnegative weight w_i. Define clause i to be

$$\mathcal{C}_i = \bigvee_{j=1}^{n_i} l_{ij},$$

where n_i is the number of literals in clause $\mathcal{C}_i$, and the literals $l_{ij} \in \{x_i, \bar{x}_i \mid i = 1, \ldots, n\}$. In the weighted *Maximum Satisfiability Problem* (MAX-SAT), one is to determine the assignment of truth values to the n variables that maximizes the sum of the weights of the satisfied clauses. Note that the classical Satisfiability Problem (SAT) is a special case of the MAX-SAT in which all clauses have unit weight and one wants to decide if there is a truth assignment of total weight m.

MAX-SAT is known to be NP-complete [6] even when each clause contains exactly two literals (MAX-2SAT). Therefore, it is unlikely that any polynomial time algorithm exists that can optimally solve MAX-SAT.

Let F^* be the optimal solution of a MAX-SAT problem. Then, an ϵ-approximation algorithm is an algorithm that produces, in polynomial time, a solution F of the MAX-SAT problem such that

$$F \geq \epsilon\, F^*, \text{where } 0 < \epsilon < 1.$$

Johnson [8] described the first ϵ-approximation algorithm for MAX-SAT with $\epsilon = \frac{1}{2}$. When each clause has at least k literals, then the algorithm becomes a $(1 - \frac{1}{2^k})$-approximation. Currently, the best ϵ-approximation algorithm for the MAX-SAT is due to Asano, Ono, and Hirata [2], with $\epsilon = .76544$. In an extension of the

Date: March 1998.

Key words and phrases. MAX-SAT problem, Satisfiability, approximate solution, heuristic, GRASP, computer implementation, Fortran subroutines.

Research partially supported by DIMACS and NSF.

```
procedure grasp(MaxIter,RandomSeed)
1    BestWeight = 0;
2    do k = 1, ... , MaxIter →
3         x = ConstructGreedyRandomizedSoln(RandomSeed);
4         x = LocalSearch(x);
5         if BestWeight < w(x)  → ;
6              BestSolution = x;
7              BestWeight = w(x);
8         fi;
9    od;
10   return(BestSolution)
end grasp;
```

FIGURE 1. GRASP pseudo-code

result in [7], Feig and Goemans [4] derive a .931-approximation algorithm for MAX-2SAT. Moreover, it is known that there exists a constant $c < 1$ such that no c-approximation algorithm exists for MAX-SAT and MAX-2SAT, unless $P = NP$ [1]. It is known [3] that $c = \frac{77}{80}$ for MAX-SAT and that $c = \frac{95}{96}$ for MAX-2SAT.

The conclusion that can be drawn from the above discussion of ϵ-approximation algorithms is that although the theoretical bound for approximation algorithms for the MAX-2SAT has been approached, this is not the case for the general MAX-SAT, justifying the investigation of heuristic algorithms for solving it. One such heuristic, a greedy randomized adaptive search procedure (GRASP) [5], was proposed by Resende, Pitsoulis, and Pardalos [13]. In this paper, we describe Fortran subroutines used to implement this heuristic.

The paper is organized as follows. In Section 2, we provide an overview of the GRASP, including the construction and local search phases. The design and implementation of the algorithm in Fortran is discussed in Section 3 and the usage of the subroutines is described in Section 4. In Section 5, we report on computational results on weighted MAX-SAT instances derived from a set of standard SAT test problems with randomly generated weights. We show the tradeoff between running time and solution quality. Concluding remarks are made in Section 6.

2. GRASP FOR THE WEIGHTED MAX-SAT

In this section, we following closely the description of the GRASP for MAX-SAT given in [13]. The GRASP implemented by the Fortran subroutines in this paper differs slightly from the GRASP in [13]. We point out the differences in this section.

Figure 1 shows a GRASP in pseudo-code. A feasible solution to a MAX-SAT instance is described by $x \in \{0,1\}^n$ and $w(x)$ is the sum of the weights of the clauses satisfied by x. The value of the best solution found is initialized in line 1. The GRASP iterations are repeated in lines 2 through 9. Each GRASP iteration has a construction phase (line 3), in which a truth assignment is produced, and a local search phase (line 4), which attempts to improve upon the constructed assignment. If necessary, the solution is updated in lines 5 through 8. The GRASP returns the best solution found.

We describe in detail the ingredients of the GRASP, i.e. the construction and local search phases.

```
procedure ConstructGreedyRandomizedSoln(RandomSeed)
1    α = GenerateRCLParameter(RandomSeed);
2    do k = 1, ... , n →
3        MakeRCL(α);
4        s = SelectIndex(RandomSeed);
5        AssignVariable(s, x);
6        AdaptGreedyFunction(s);
7    od;
8    return x;
end ConstructGreedyRandomizedSoln;
```

FIGURE 2. GRASP construction phase pseudo-code

2.1. **Construction phase.** The construction phase of a GRASP builds a solution, around whose neighborhood a local search is carried out in the local search phase, producing a locally optimal solution. This construction phase solution is built, one element at a time, guided by a greedy function and randomization. Figure 2 describes in pseudo-code a GRASP construction phase. Since in the MAX-SAT problem there are n variables to be assigned, each construction phase consists of n iterations. In `GenerateRCLParameter`, the restricted candidate list parameter α is generated. We discuss this in more detail later in this subsection. In `MakeRCL` the restricted candidate list of assignments is set up. The index of the next variable to be assigned is chosen in `SelectIndex`. The variable selected is assigned a truth value in `AssignVariable`. In `AdaptGreedyFunction` the greedy function that guides the construction phase is changed to reflect the assignment just made. To describe these steps in detail, we need some definitions. Let $N = \{1, 2, \ldots, n\}$ and $M = \{1, 2, \ldots, m\}$ be sets of indices for the set of variables and clauses, respectively. Solutions are constructed by setting one variable at a time to either 1 (`true`) or 0 (`false`). Therefore, to define a restricted candidate list, we have 2 potential candidates for each yet-unassigned variable: assign the variable to 1 or assign the variable to 0.

We now define the adaptive greedy function. The idea behind the greedy function is to maximize the total weight of yet-unsatisfied clauses that become satisfied after the assignment of each construction phase iteration. For $i \in N$, let Γ_i^+ be the set of unassigned clauses that would become satisfied if variable x_i were to be set to `true`. Likewise, let Γ_i^- be the set of unassigned clauses that would become satisfied if variable x_i were to be set to `false`. Define

$$\gamma_i^+ = \sum_{j \in \Gamma_i^+} w_j \text{ and } \gamma_i^- = \sum_{j \in \Gamma_i^-} w_j.$$

The greedy choice is to select the variable x_k with the largest γ_k^+ or γ_k^- value and set it to the corresponding truth value. If $\gamma_k^+ > \gamma_k^-$, then the assignment $x_k = 1$ is made, else $x_k = 0$. Note that with every assignment made, the sets Γ_i^+ and Γ_i^- change for all i such that x_i is not assigned a truth value, to reflect the new assignment. This consequently changes the values of γ_i^+ and γ_i^-, characterizing the adaptive component of the heuristic.

Next, we discuss the restriction mechanism for the restricted candidate list (RCL) used in this paper. The RCL is set up in `MakeRCL` of the pseudo-code of Figure 2.

```
procedure AdaptGreedyFunction(s)
1    if s > 0 →
2        for j ∈ Γ_s^+ →
3            for k ∈ L_j (k ≠ j) →
4                if x_k is unnegated in clause j →
5                    Γ_k^+ = Γ_k^+ − {j}; γ_k^+ = γ_k^+ − w_j;
6                fi;
7                if x_k is negated in clause j →
8                    Γ_k^− = Γ_k^− − {j}; γ_k^− = γ_k^− − w_j;
9                fi;
10           rof;
11       rof;
12       Γ_s^+ = ∅; Γ_s^− = ∅;
13       γ_s^+ = 0; γ_s^− = 0;
14   fi;
15   if s < 0 →
16       for j ∈ Γ_{−s}^− →
17           for k ∈ L_j (k ≠ j) →
18               if x_k is unnegated in clause j →
19                   Γ_k^+ = Γ_k^+ − {j}; γ_k^+ = γ_k^+ − w_j;
20               fi;
21               if x_k is negated in clause j →
22                   Γ_k^− = Γ_k^− − {j}; γ_k^− = γ_k^− − w_j;
23               fi;
24           rof;
25       rof;
26       Γ_{−s}^+ = ∅; Γ_{−s}^+ = ∅;
27       γ_{−s}^+ = 0; γ_{−s}^− = 0;
28   fi;
29   return
end AdaptGreedyFunction;
```

FIGURE 3. `AdaptGreedyFunction` pseudo-code

A value restriction mechanism is used. It imposes a parameter based *achievement level*, that a candidate has to satisfy to be included in the RCL. In this way we ensure that a random selection will be made among the best candidates in any given assignment. Let

$$\gamma^* = \max\{\gamma_i^+, \gamma_i^- \mid x_i \text{ yet unassigned}\}$$

and

$$\gamma_* = \min\{\gamma_i^+, \gamma_i^- \mid x_i \text{ yet unassigned}\}.$$

Let α $(0 \leq \alpha \leq 1)$ be the restricted candidate parameter. A candidate $x_i = \texttt{true}$ is inserted into the RCL if $\gamma_i^+ \geq \gamma_* + \alpha \cdot (\gamma^* - \gamma_*)$. Likewise, a candidate $x_i = \texttt{false}$ is inserted if $\gamma_i^- \geq \gamma_* + \alpha \cdot (\gamma^* - \gamma_*)$.

In the implementation used to produce the computational results given in [13], the RCL parameter α was fixed to 0.5 for all GRASP iterations. A more robust

```
procedure LocalSearch(x)
1    BestSolutionFound = C(x);
2    GenerateGains(x,G,0);
3    G_k = max{G_i | i = 1, ... , n} ;
4    for G_k ≠ 0 →
5         Flip value of x_k;
6         GenerateGains(x,G,k);
7    rof;
8    return;
end LocalSearch;
```

FIGURE 4. The local search procedure in pseudo-code

implementation uses an RCL parameter that is not fixed. Prais and Ribeiro [12] describe a reactive GRASP, which dynamically changes the value of the RCL parameter. In the implementation described in this paper, we select a new value of α for each GRASP iteration. The parameter is selected, at random, from the uniform distribution $U[0,1]$. This is done in `GenerateRCLParameter`. The added benefit of this selection scheme is that this GRASP converges asymptotically to the global optimum [11], since every solution in the feasible space has positive probability to be generated by the construction phase.

Once the RCL is set up, a candidate from the list must be selected and made part of the solution being constructed. `SelectIndex` selects at random the index s from the RCL. In `AssignVariable`, the assignment is made, i.e. $x_s =$ `true` if $s > 0$ or $x_s =$ `false` if $s < 0$.

The greedy function is changed in `AdaptGreedyFunction` to reflect the assignment made in `AssignVariable`. This requires that some of the sets Γ_i^+, Γ_i^-, as well as the γ_i^+ and γ_i^-, be updated. There are two cases, as described in Figure 3. If the variable just assigned was set to `true` then Γ^+, Γ^-, γ^+ and γ^- are updated in lines 5, 8, 12, and 13. If the variable just assigned was set to `false` then Γ^+, Γ^-, γ^+ and γ^- are updated in lines 19, 22, 26, and 27.

2.2. **Local search phase.** The GRASP construction phase described in Subsection 2.1 computes a feasible truth assignment that is not necessarily locally optimal with respect some neighborhood structure. Consequently, local search can be applied with the objective of finding a locally optimal solution that may be better than the constructed solution. In fact, the main purpose of the construction phase is to produce a diverse set of good initial solutions for the local search.

To define the local search procedure, some preliminary definitions have to be made. Given a truth assignment $x \in \{0,1\}^n$, define the *1-flip neighborhood* $N(x)$ to be the set of all vectors $y \in \{0,1\}^n$ such that $\| x - y \|_2 = 1$. If x is interpreted as a vertex of the n-dimensional unit hypercube, then its neighborhood consists of the n vertices adjacent to x. If we denote by $w(x)$ the total weight of the clauses satisfied by the truth assignment x, then the truth assignment x is a *local maximum* if and only if $w(x) \geq w(y)$, for all $y \in N(x)$. Starting with a truth assignment x, the local search finds the local maximum y in $N(x)$. If $y \neq x$, it sets $x = y$. This process is repeated until no further improvement is possible.

Note that a straightforward implementation of the local search procedure described above, would require n function evaluations to compute a local maximum, where for a given assignment, each function evaluation computes the total sum of the weights of the satisfied clauses. Moreover, this process is repeated until the local maximum is the initial solution itself, a process that could result in an exponential number of computational steps [9, 10]. We can, however, exploit the structure of the neighborhood to reduce the computational effort.

Given an initial solution x define G_i to be the gain in total weight resulting from flipping variable x_i in x, for all i. Let $G_k = \max\{G_i \mid i \in N\}$. If $G_k = 0$ then x is the local maximum and local search ends. Otherwise, the truth assignment resulting from flipping x_k in x, is a local maximum, and hence we only need to update the G_i values such that the variable x_i occurs in a clause in which variable x_k occurs (since the remaining G_i values do not change in the new truth assignment). Upon updating the G_i values we repeat the same process, until $G_k = 0$ where the local search procedure is terminated. The procedure is described in the pseudo-code in Figure 4. Given a truth assignment x and an index k that corresponds to the variable x_k that is flipped, procedure `GenerateGains` is used to update the G_i values returned in an array G. Note that, in line 2, we pass $k = 0$ to the procedure, since initially all the G_i values must be generated (by convention variable x_0 occurs in all clauses). In lines 4 through 7, the procedure finds a local maximum. The value of the local maximum is saved in line 8.

3. Design and Implementation

In this section, we discuss several issues related to the design and implementation of the Fortran subroutines. We describe the design features of the code, the distribution, as well as data structures implemented. Usage of the code is described later, in Section 4.

3.1. **Code design and distribution.**

We followed a few guidelines in the design of the code.

- The subroutines are written in ISO Standard Fortran 77.
- There are no COMMON blocks in the subroutines. All communication between subroutines is done by parameters.
- The main optimization subroutine is completely controlled by the calling program. The calling program specifies maximum instance size, algorithm control parameters, printing control parameters, and does exception handling upon receiving an error condition. The user need not modify a single line of code in the optimization subroutines.
- A minimum description of the problem instance is passed to the main optimization subroutine. The optimization subroutine checks for memory availability, consistency of the input data, as well as consistency of algorithm and program control parameters.
- Because the data structure needed to implement the neighborhood data structure for reduced updating in the local search procedure requires $4(m + n^2)$ bytes (where m is the number clauses and n is the number of variables), two versions of the code are distributed: a large memory version (`gmsatl`), which implements the neighborhood data structure and a small memory version (`gmsats`), which does not use this data structure. `gmsats` is less efficient than

`gmsatl` but requires less storage. Both optimizers produce identical solutions for the same input.

The distribution is made up of 8 files.

- A `Read.Me` file provides basic information on the code.
- A `Makefile` compiles and links the two versions of the code under the UNIX Operating System.
- The large memory version is made up of files `driverl.f` and `gmsatl.f`.
- The small memory version consists of `drivers.f` and `gmsats.f`.
- A sample input file `sample.sat`, readable by `readp.f`, is provided, as well as one output `sample.out`, produced by the code for this sample input.

The files `driver[ls].f` contain parameter and array definitions, a driver program, and subroutines to read the problem instance (`readp`), print out error messages (`errmsg`), print input summary (`iniprt`), and output the solution found (`outsol`). The driver program also makes the call to the optimizer. Figure 5 illustrates calling sequence for `driverl.f`. The small memory version `drivers.f` differs from `driverl.f` only in the parameter and array definitions and the call to the optimizer, which in the small memory version is subroutine `gmsats`.

Files `gmsat[ls].f` contain the following subroutines:

- Subroutine `chkinp` checks for consistency in storage allocation, algorithm and program control parameters, and input data specification. A nonzero error condition is returned if an error is identified.
- Subroutines `mkds[ls]` take the input (that was minimally specified) and creates the data structures needed by the program. Subroutine `mkdsl` and `mkdss` differ in that `mkdsl` creates the neighborhood data structure used for reduced updating in the local search procedure, while `mkdss` does not.
- Subroutine `randp` is the pseudo random number generator of Schrage [14].
- The initial restricted candidate list (RCL) at each GRASP iteration is constructed by subroutine `mkrcl`.
- The GRASP construction phase is implemented in subroutine `build`, which itself calls subroutine `updrcl` that updates the RCL as each candidate variable is assigned a truth value.
- The GRASP local search phase is implemented in two flavors: small memory (`locals`) and large memory (`locall`). `locals` is called by subroutine `gmsats`, while `locall` is called by `gmsatl`.
- If an improved assignment is found, subroutine `savsol` is called to record the assignment, as well as the total weight of the assignment and the pseudo random number generator seed at the start of the GRASP iteration in which the improved solution was found.
- Subroutine `copyi4` makes a copy of an integer array.

3.2. **Data structures.** We now describe the data structures used in the program. We begin by describing the data structure used to represent the instance, and then describe additional data structures used to implement the GRASP.

The instance is represented by three integer parameters and three integer arrays. Parameters `n`, `m`, and `numlit` are, respectively, the number of variables, clauses, and literals in the MAX-SAT instance. Array `w` of dimension `m` contains the clause weights. Element `w(`i`)` is the weight of clause i. The clauses are stored in the integer arrays `headc` and `lit`. Array `lit` stores the variable indices of the literals. The k-th

```
      program driverl
c     -------------------------------------------
c     parameter and array definitions
c     -------------------------------------------
      integer ···
c     -------------------------------------------
c     input problem
c     -------------------------------------------
      call readp(···)
c     -------------------------------------------
c     if input not successful, print error message
c     -------------------------------------------
      if ( errcnd .gt.  0 ) then
           call errmsg(errcnd,iout)
      endif
c     -------------------------------------------
c     print input report
c     -------------------------------------------
      call iniprt(···)
c     -------------------------------------------
c     run optimizer
c     -------------------------------------------
      call gmsatl(···)
c     -------------------------------------------
c     print solution, or error message
c     -------------------------------------------
      call errmsg(errcnd,iout)
      if ( errcnd .eq.  0 ) then
           call outsol(···)
      endif
c     -------------------------------------------
      stop
      end
```

FIGURE 5. driver program skeleton (large memory version)

literal x_j is represented by `lit(`k`)` $= j > 0$, while the k-th literal $\overline{x}_j$ is represented by `lit(`k`)` $= -j < 0$. All literals in a given clause are stored consecutively in array `lit`. The starting and ending position in array `lit` of the literals in a specific clause are stored in array `headc`. The first literal of clause i is stored in position `headc(`i`)` of array `lit`, while the last literal is in position `headc(`$i+1$`)`-1. Arrays `lit` and `headc` have dimensions `numlit` and `m` $+1$, respectively.

As an example of the instance representation, consider the following MAX-SAT problem with 3 clauses and 5 variables:

$$\begin{aligned} w(1) &= 100; \quad x_1 \vee \overline{x}_3 \vee \overline{x}_5 \\ w(2) &= 500; \quad x_2 \vee \overline{x}_4 \\ w(3) &= 700; \quad \overline{x}_1 \vee x_3 \vee x_5. \end{aligned}$$

The parameters and arrays to represent this instance are:

$$\begin{aligned} \texttt{n} &= 5 \\ \texttt{m} &= 3 \\ \texttt{numlit} &= 8 \\ \texttt{w} &= [100, 500, 700] \\ \texttt{headc} &= [1, 4, 6, 9] \\ \texttt{lit} &= [1, -3, -5, 2, -4, -1, 3, 5]. \end{aligned}$$

To implement the GRASP efficiently, we use the following data structure.

Each MAX-SAT variable has associated with it two linked lists. The first links all literals of that variable that occur unnegated, while the second links all literals of that variable that occur negated. These lists are represented by the integer arrays `headp` (of size `n`), `nextp` (of size `numlit`), `headm` (of size `n`), and `nextm` (of size `numlit`). Element `headp(`i`)` (`headm(`i`)`) points to the first element in the list of unnegated (negated) literals of variable i. Element `nextp(`k`)` (`nextm(`k`)`) points to the next element after literal `lit(`k`)` in the list of unnegated (negated) literals of variable i.

To access the clause in which a literal occurs, array `clause` is used. Element `clause(`k`)` indicates the clause number of the literal `lit(`k`)`.

The adaptive component of the GRASP makes use of the additional weight gained by yet unsatisfied clauses that would become satisfied if a particular assignment were to be made. This is implemented with four arrays. The i-th element of array `minus0` (`plus0`) contains the weight gained by clauses that would become satisfied if variable x_i were to be set to 1 (0). These two arrays are setup once and do not change. They are copied at the start of each GRASP iteration to arrays `minus` and `plus` which are updated as the GRASP construction phase proceeds. At any stage in the construction phase, the i-th element of array `minus` (`plus`) contains the additional weight gained by yet unsatisfied clauses that would become satisfied if variable x_i were to be set to 1 (0).

Array `satcl` (of size `m`) indicates with a 0 or 1 whether a clause is satisfied or not. `satcl(`i`)` $= 1$ indicates that clause i is satisfied, while `satcl(`i`)` $= 0$ indicates that clause i is not yet satisfied.

The restricted candidate list is represented by array `rcl` (of maximum size 2 `n`). Element `rcl(`k`)` $= j > 0$ represents the assignment $x_j = 1$, while `rcl(`k`)` $= -j < 0$ represents the assignment $x_j = 0$. The number of elements in the RCL is given by `nrcl`.

The current solution is kept in array `x` (of size `n`). Element `x(`j`)` $= 1$ represents the assignment $x_j = 1$, while `x(`j`)` $= -1$ represents the assignment $x_j = 0$. The incumbent solution is stored in array `xopt`.

Both local search versions use array `gainx` (of size `n`) to store the weight gain achieved by flipping the assignment of a variable. Element `gainx(`j`)` represents the weight gain achieved by flipping the value of variable x_j, i.e. making $x_j = 1$ if $x_j = 0$, or making $x_j = 0$ if $x_j = 1$.

The large memory version of local search also uses the neighborhood data structure. This data structure indicates, for each variable index i, the set of variable indices that appear in at least one of the clauses where variable x_i appears. The indices are stored consecutively in an integer array `xx` (of size at most `n` $\times$ `n`),

with integer array `xptr` (of size `n` +1) used to indicate the start and end of the neighborhoods of each variable.

Suppose, once again, we have the following clauses in the MAX-SAT instance:

$$x_1 \vee \overline{x}_3 \vee \overline{x}_5$$
$$x_2 \vee \overline{x}_4$$
$$\overline{x}_1 \vee x_3 \vee x_5.$$

Then, `xx` and `xptr` will be

$$\texttt{xx} = [1, 3, 5, 2, 4, 3, 1, 5, 4, 2, 5, 1, 3]$$
$$\texttt{xptr} = [1, 4, 6, 9, 11].$$

Note that, by convension, each x_i is in the neighborhood of itself. In the local search, each time a flip is done, the potential weight gain (`gainx`) of the other variables must be recomputed. The neighborhood data structure allows us to restrict our updates only to variables that are in the neighborhood of the flipped variable, thus improving the efficiency of the updates.

4. Usage

In this section, we describe usage of the subroutine. We show to compile the codes, run the codes with the driver programs, and use the optimizers without the driver programs that are provided.

The memory requirements to store the arrays in the small and large memory versions are, respectively, $4(11n + 2m + 4l)$ and $4(n^2 + 12n + 2m + 4l)$, where n is the number of variables, m is the number of clauses, and l is the number of literals.

4.1. **Making an executable.**

As described in Section 3, the distribution contains 8 files, 5 of which are needed to prepare an executable: `Makefile`, `drivel.f`, `gmsatl.f`, `drives.f`, and `gmsats.f`. Before compiling the two programs, the user may have to adjust a few parameters in the driver programs. We describe this later in this section.

After editing the `Makefile` and adjusting it to the environment that the executable will run on, simply type:

```
make gmsatl
```

to produce a large memory version executable (`gmsatl`), or

```
make gmsats
```

to produce a small memory version executable (`gmsats`).

To test the executable, use the input data file `sample.sat` in the distribution and type

```
gmsatl < sample.sat
```

to run the large memory version, or type

```
gmsats < sample.sat
```

to run the small memory version. With either case, the output produced should be exactly what is in the distribution file `sample.out`.

```
      program driverl
      ...
c     ---------------------------------------------
      parameter (nmax=1000,mmax=5000,lmax=20000)
      ...
      parameter (in=5,iout=6)
c     ---------------------------------------------
      ...
      look4=2147483647
      niter=100
      prttyp=1
      seed=177890629
      ...
c     ---------------------------------------------
      stop
      end
```

FIGURE 6. Parameter adjusting in driver program

4.2. **Using the driver programs.** The subroutines can be run with the driver programs provided in the distribution. Before compiling the code, the user should edit the corresponding driver file and adjust the lines in Figure 6. The integer parameters `nmax`, `mmax`, and `lmax` are, respectively, the maximum number of variables, maximum number of clauses, and maximum number of literals that the code can handle. Parameters `in` and `iout` are, respectively, the Fortran input and output device numbers.

Parameters `look4`, `niter`, `prttyp`, and `seed` control the optimizer and the code. If the GRASP finds a truth assignment at least as large as `look4`, then optimization is concluded and control returns to the calling program. The optimizer executes at most `niter` GRASP iterations. Parameter `prttyp` controls what is printed by the optimizer. If `prttyp` $= 0$, subroutine `gmsatl` (`gmsats`) is silent, i.e. prints nothing. If `prttyp` $= 1$, subroutine `gmsatl` (`gmsats`) prints only iteration summaries for GRASP iterations in which the incumbent is improved. Finally, if `prttyp` $= 2$, subroutine `gmsatl` (`gmsats`) prints iteration summaries for all GRASP iterations. The pseudo random number generator seed (`seed`) is an integer between 1 and $2^{31} - 1$. Different seeds will lead to different GRASP runs. On multiprocessor computers, the user may want to run multiple copies of GRASP, each running on a different processor. To do this, all that is needed are different seeds to each GRASP copy such that the seed sequences are nonoverlapping.

Subroutine `readp` reads the instance given as follows. Input is not formatted. In line 1, `n` (number of variables) and `m` (number of clauses), are read. Then, for each clause i, the input file specifies the number of literals in that clause `litcl`, the clause weight (`w(`i`)`), and the literal indices with a negative sign if the literal is negated (`lit`). Consider again our example:

$$\begin{aligned} w(1) &= 100; \quad x_1 \vee \overline{x}_3 \vee \overline{x}_5 \\ w(2) &= 500; \quad x_2 \vee \overline{x}_4 \\ w(3) &= 700; \quad \overline{x}_1 \vee x_3 \vee x_5. \end{aligned}$$

The input file for this example is given in Figure 7.

```
5 3
3   100  1 -3 -5
2   500  2 -4
3   700  -1 3 5
```

FIGURE 7. Sample input file for `readp`

Figure 8 illustrates the ouput produced by running the code with print option `prttyp` = 1, stopping parameter `look4` = 420238, and maximum iteration parameter `niter = 100000`, and random number generator seed parameter `seed = 1778090629` on problem instance `jnh201` [13].

4.3. **Calling the optimizers.** In some situations, the user may not want to use the drivers provided, but rather call the optimizers from the user's own program. To do this, we specify in this section, what parameter and array definitions need to be made in the calling program, the calling sequence to the optimizer, and provide the list of possible error conditions.

For both subroutines `gmsatl` and `gmsats`, the following integer array dimension parameters must be defined:

```
integer    lmax, mmax, mp1max, nmax, nt2max
parameter  (lmax=10000, mmax=100, nmax=100)
parameter  (mp1max=mmax+1, nt2max=nmax*2)
```

where the values of `lmax`, `nmax`, and `mmax` should be set according to the size of the MAX-SAT instance to be solved. Furthermore, for subroutine `gmsatl`, the following additional parameter is needed:

```
integer    np1max, ntnmax
parameter  (np1max=nmax+1, ntnmax=nmax*nmax)
```

Output device parameter `iout` must be specified:

```
integer    iout
parameter  (iout=6)
```

where, if required, the value "6" can be changed.

The following integer parameters that are either input to or output from `gmsatl` (`gmsats`) must be defined:

```
integer    bestv, bests, errcnd, iter, look4, m, n, niter
integer    numlit, prttyp, seed
```

The integer arrays

```
integer    clause(lmax), gainx(nmax), headc(mp1max)
integer    headm(nmax), headp(nmax), lit(lmax)
integer    minus(nmax), minus0(nmax), nextm(lmax)
integer    nextp(lmax), plus(nmax), plus0(nmax)
integer    rcl(nt2max), satcl(mmax), w(mmax)
integer    x(nmax), xopt(np1max)
```

must be defined for both `gmsatl` and `gmsats`. In addition, for subroutine `gmsatl`, the following arrays are required:

```
integer    xptr(np1max), xopt(ntnmax)
```

```
       GRASP/MAX-SAT:

number of variables :        100
  number of clauses :        800
 number of literals :       4154
   iteration limit  :     100000
       initial seed : 1778090629
       print option :          1
           look for :     394238

     iter alpha  ph-1-soln  ph-2-soln  best-soln
        1  0.99     390251     393033     393033 <<<<improvement
        6  0.56     387156     393061     393061 <<<<improvement
       11  0.63     388393     393187     393187 <<<<improvement
       12  0.13     378433     393498     393498 <<<<improvement
       23  0.17     382773     394121     394121 <<<<improvement
     1641  0.26     381044     394145     394145 <<<<improvement
     9236  0.13     380218     394155     394155 <<<<improvement
    10937  0.19     383461     394227     394227 <<<<improvement
    53158  0.36     386673     394238     394238 <<<<improvement

Execution terminated with no error.

         GRASP/output :

 number of iterations :        53158
 seed  best iteration :   1579194944
        best solution :       394238

     truth assignment :  0 1 1 0 0 0 0 1 0 0 0 0 1 0 0 1 0 1 1 0 1 1
     truth assignment :  0 1 0 1 1 1 0 0 0 1 1 0 1 0 0 0 1 0 0 1 1 0
     truth assignment :  1 0 0 0 1 0 0 1 0 0 0 0 0 1 0 1 1 0 1 1 0 0 0
     truth assignment :  0 0 0 0 0 0 1 0 1 1 0 0 0 1 0 0 1 1 0 0 1 0 1
     truth assignment :  0 0 1 1 1 1 1 0 1 0 0 1 0
```

FIGURE 8. Sample output file for `jnh12`

Before calling the optimizer, the problem instance must be placed in the variables and arrays `n`, `m`, `numlit`, `headc`, and `lit`, as was described in Subsection 3.2, and control parameters `look4`, `niter`, `prttyp`, and `seed` must be set, as described in Subsection 4.2.

The calling sequence to the large memory version optimizer (`gmsatl`) is

```
      call gmsatl ( bestv, bests, clause, gainx, headc, headm,
     +              headp, iout, iter, lit, look4, lmax,
     +              m, minus, minus0, mmax, n, nextm,
     +              nextp, niter, nmax, numlit, plus, plus0,
     +              prttyp, rcl, satcl, seed, w, x,
```

TABLE 1. Error conditions

errcnd	description
0	No error.
1	Error in n. $n < 1$ or $n >$ nmax.
2	Error in m. $m < 1$ or $m >$ mmax.
3	Error in numlit. numlit < 1 or numlit $>$ lmax.
4	Error in weight array w. w$(i) < 0$ or w$(i) > 2^{31} - 1$.
5	Error in stopping parameter look4. look4 < 1 or look4 $> 2^{31} - 1$.
6	Error in maximum iteration parameter niter. niter < 1 or niter $> 2^{31} - 1$.
7	Error in print option parameter prttyp. prttyp $\neq 0, 1$ or 2.
8	Error in random number generator seed. seed < 1 or seed $> 2^{31} - 1$.
9	Error in input. headc$(i) >$ lmax.
10	Error in input. headc$(i) =$ headc$(i+1)$.
11	Error in input. headc$(i) >$ headc$(i+1)$.
12	Error in input. lit$(i) < -n$ or lit$(i) > n$.

```
+              xopt, xptr, xx )
```

while the calling sequence to the small memory version (gmsats) is

```
 call gmsats ( bestv, bests, clause, gainx, headc, headm,
+              headp, iout, iter, lit, look4, lmax,
+              m, minus, minus0, mmax, n, nextm,
+              nextp, niter, nmax, numlit, plus, plus0,
+              prttyp, rcl, satcl, seed, w, x,
+              xopt )
```

If no inconsistency is detected by the optimizer, an error condition code errcnd $=$ 0 is returned and, the best found assignment is returned in the 0-1 array xopt, with the total weight of the best found in bestv, the iteration in which the best solution was found in iter, and the seed at the start of the iteration in which the best solution was found in bests.

If an error is found, the error condition parameter returned errcnd > 0. Table 4.3 lists the error conditions.

5. Computational results

In this section, we report on experiments using the Fortran subroutines. We limit our experiments to the instances reported in [13] for which, in that paper, no optimal solution was found in 100,000 GRASP iterations. These instances (jnh1, jnh10, jnh11, jnh12, jnh201, jnh202, jnh212, jnh304, jnh305, and jnh306) are available at http://www.research.att.com/~mgcr/data/maxsat.tar.gz. They have 100 variables and 800 clauses (jnh1, jnh10, jnh11, and jnh12), 100 variables and 850 clauses (jnh201, jnh202, and jnh212), and 100 variables and 900 clauses (jnh304, jnh305, and jnh306). All instances have known optimal solutions.

The experiments were done on a Silicon Graphics Challenge computer with 20 196MHz MIPS R10000 processors and 6.8 Gbytes of memory. The codes were compiled on the SGI Fortran compiler using flags -O3 -64 -static -u. Running times were measured using the system call etime. All times reported are user times.

For each instance, we ran the code in parallel on 10 processors, using initial seeds such that the seed sequences in each processor are disjoint from each other. To do this, we ran the pseudo random number generator [14] for $2^{31} - 1$ iterations, recording the seeds every $(2^{31} - 1)/10$ iterations. The initial seeds

TABLE 2. Quality of best GRASP solution as a function of iterations

	GRASP iterations				
name	1000	100,000	1,000,000	10,000,000	optimal
jnh1	420410	420739	420819	420925	420925
jnh10	419754	420357	420754	420758	420840
jnh11	419717	420516	420740	420753	420753
jnh12	419921	420871	420920	420925	420925
jnh201	393905	394222	394238	394238	394238
jnh202	393483	393870	394044	394170	394170
jnh212	393414	394006	394227	394227	394238
jnh304	443501	444125	444533	444533	444533
jnh305	442696	443815	444112	444112	444112
jnh306	444145	444692	444774	444838	444838

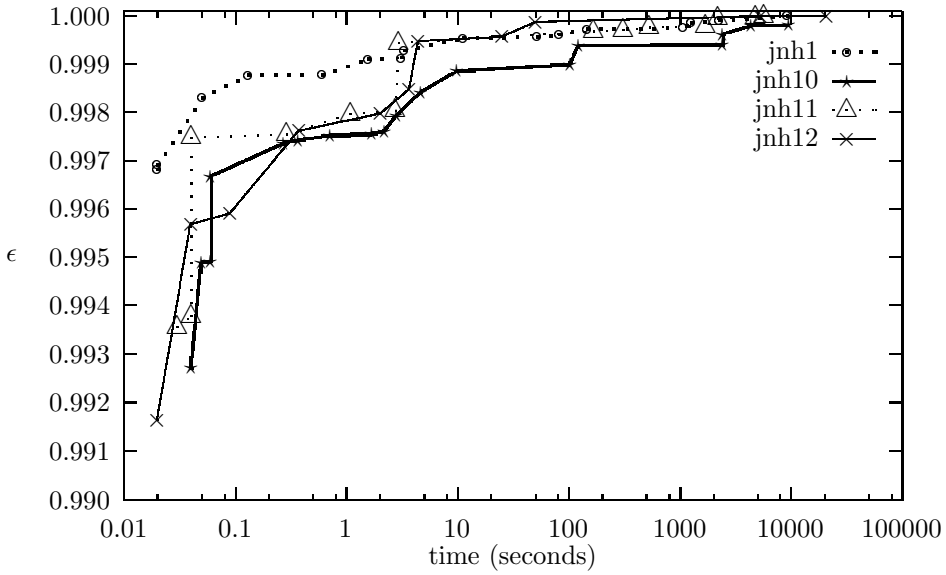

FIGURE 9. ϵ-approximate solution as a function of time: `jnh1`, `jnh10`, `jnh11`, and `jnh12`

used are 1778090629, 43183541, 1178123378, 1211176980, 1962386846, 1265238465, 1930924806, 470197762, 1701589954, and 220430966. For each run, the number of GRASP iterations was set to `niter = 1000000`.

The experiments were done using the large memory version (`gmsatl`) of the Fortran subroutines.

In 8 of the 10 instances tested, the GRASP found the optimal solution. In the two instances were optimal solutions were not found, the percentage relative errors were 0.01948% (for `jnh10`) and 0.00279% (for `jnh212`), which for most practical purposes are as good as optimal. Perhaps more importantly, in all runs ϵ-approximate solutions with $\epsilon > .99$ were found in less than 0.05 seconds.

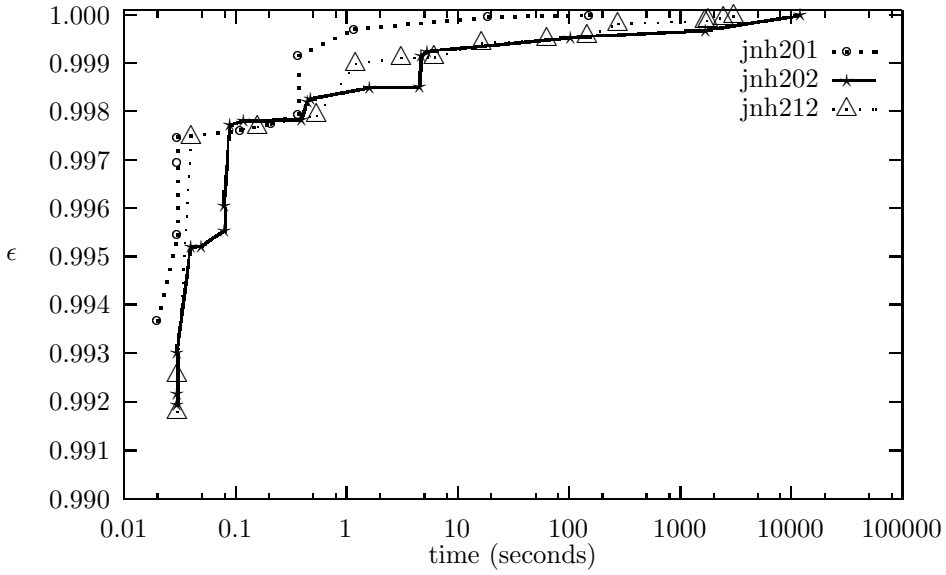

FIGURE 10. ϵ-approximate solution as a function of time: `jnh201`, `jnh202`, and `jnh212`

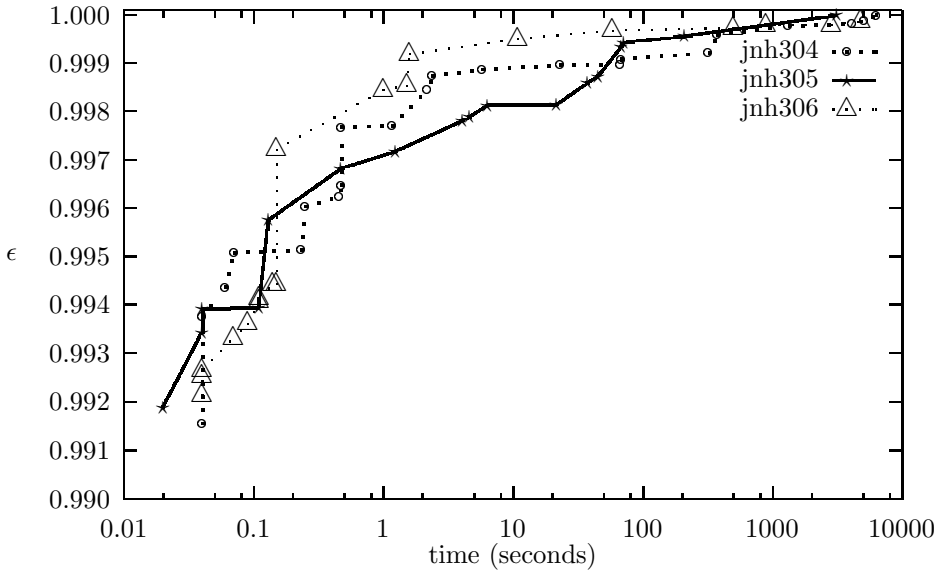

FIGURE 11. ϵ-approximate solution as a function of time: `jnh304`, `jnh305`, and `jnh306`

Table 5 illustrates the quality of the best solution found as a number of GRASP iterations (added over all processors) and Figures 9–11 illustrate the performance of the subroutines. The plots on those figures, show how the ratio ϵ of the best solution found so far to the optimal solution improves with running time. Observe that the subroutines produced almost optimal solutions in a fraction of a second and that all runs produced solutions with $\epsilon > 0.999$.

6. Concluding remarks and discussion

In this paper, we describe a set of Fortran subroutines for finding approximate solutions of weighted MAX-SAT instances using GRASP. Two versions of the subroutines are distributed. The first version, the large memory version, makes use of a neighborhood data structure to speed up the local search phase of GRASP. The second version, which does not use this data structure, is more memory efficient, but is less time efficient.

The subroutines can be used with a driver program that is provided in the distribution or can be called from any other subroutine. We describe how to use the subroutines in detail.

To illustrate the effectiveness of the subroutines, we report on computational experience with the large memory version of the code. Our experiments show that the subroutines can produce high-quality solutions in a fraction of a second and optimal or almost optimal solutions if allowed to run longer.

The codes can be downloaded from `http://www.research.att.com/~mgcr-/src/maxsat.tar.gz`.

References

[1] S. Arora, C. Lund, R. Motwani, M. Sudan, and M. Szegedy. Proof verification and hardness of approximation problems. In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 14–23, 1992.

[2] T. Asano, T. Ono, and T. Hirata. Approximation algorithms for the maximum satisfiability problem. *Nordic Journal of Computing*, 3:388–404, 1996.

[3] M. Bellare, O. Goldreich, and M. Sudan. Free bits, pcp and non-approximability — towards tight results. *Unpublished manuscript*, 1995.

[4] U. Feige and M.X. Goemans. Approximating the value of two proper proof systems, with applications to MAX-2SAT and MAX-DICUT. In *Proceeding of the Third Israel Symposium on Theory of Computing and Systems*, pages 182–189, 1995.

[5] T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–133, 1995.

[6] M.R. Garey and D.S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Company, New York, 1979.

[7] M.X. Goemans and D.P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of Association for Computing Machinery*, 42(6):1115–1145, 1995.

[8] D.S. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.

[9] D.S. Johnson, C.H. Papadimitriou, and M. Yannakakis. How easy is local search? *Journal of Computer and System Sciences*, 37:79–100, 1988.

[10] M.W. Krentel. The complexity of optimization problems. *Journal of Computer and System Sciences*, 36, 1988.

[11] J. Mockus, W. F. Eddy, A. Mockus, L. Mockus, and G. Reklaitis. *Bayesian Heuristic Approach to Discrete and Global Optimization*. Kluwer Academic Publishers, Dordrecht, 1997.

[12] M. Prais and C.C. Ribeiro. Reactive GRASP: An application to a matrix decomposition problem in TDMA traffic assignment. Technical report, Department of Computer Science, Catholic University of Rio de Janeiro, Rio de Janeiro, RJ 22453-900 Brazil, 1998.

[13] M.G.C. Resende, L.S. Pitsoulis, and P.M. Pardalos. Approximate solution of weighted MAX-SAT problems using GRASP. In D.-Z. Du, J. Gu, and P.M. Pardalos, editors, *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pages 393–405. American Mathematical Society, 1997.

[14] L. Schrage. A more portable Fortran random number generator. *ACM Transactions on Mathematical Software*, 5:132–138, 1979.

(M. G. C. Resende) Information Sciences Research, AT&T Labs Research, Florham Park, NJ 07932 USA.

E-mail address: mgcr@research.att.com

(L. S. Pitsoulis) Center for Applied Optimization, Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611 USA.

E-mail address: leonidas@deming.ise.ufl.edu

(P. M. Pardalos) Center for Applied Optimization, Department of Industrial and Systems Engineering, University of Florida, Gainesville, FL 32611 USA.

E-mail address: pardalos@ufl.edu